TITLE

SYSTEMS AND METHODS FOR SHARING OF EXECUTION PLANS FOR SIMILAR DATABASE STATEMENTS

INVENTORS

SANJAY KALUSKAR
NAMIT JAIN

ASSIGNEE

ORACLE CORPORATION

Atty. Dkt. No. 264/237
OID-2001-046-01

SYSTEMS AND METHODS FOR SHARING OF EXECUTION PLANS FOR SIMILAR DATABASE STATEMENTS

Field of the Invention

[01] This application relates generally to the sharing of data structures, and more particularly relates to systems and methods for sharing of execution plans for similar database statements.

Background and Summary

[02] In modern relational database management systems (RDBMS), the overhead associated with processing client requests can be significant. Cache and buffer overflow, I/O bottlenecks, wasted CPU cycle time, shared memory latch contention, network throughput, and other performance side effects often result from poor planning and untested design.

[03] To avoid these and other by-products of a poorly designed system, a client/server DBMS architecture could benefit greatly from a streamlined database statement processing system. In a typical two-tier DBMS architecture, a client issues a database statement (hereinafter illustratively referred to as a "SQL statement") to a process running on the database server through a proprietary or open-system call level interface (CLI). The server expends a great deal of its run-time resources in parsing the request, creating an execution tree, semantically analyzing the statement, and determining an optimal execution plan. These steps together constitute the compilation, or "hard parse,"

steps needed to store and create a cursor in cache memory before the server can effectively carry out a client request or return a result set.

[04] It would be highly desirable to implement a process and system for conserving system resources and reducing the overhead/expense of processing SQL statements. Conserving system resources means avoiding, where possible, the redundant processing of client SQL statements. While a DBA can accomplish some performance enhancement via run-time parameter adjustment, often the parameters available to the DBA are not tailored specifically to execution plan reuse.

[05] The systems and methods for sharing of execution plans for similar SQL statements, according to embodiments of the invention, minimize or eliminate the inherent limitations and drawbacks of current SQL processing techniques by permitting a SQL statement issued from a client to reuse a cursor pre-compiled for a similar SQL statement.

[06] In one embodiment, the systems and methods for sharing of execution plans comprises the following system components: a search engine, a cursor sharing monitor, and a compiler. The search engine searches the shared memory pool for a pre-existing cursor built from similar SQL text that matches the text of the issued SQL statement. The cursor sharing monitor invokes and monitors execution plan reuse if a match is found, or if a match is not found and the previously mentioned configurable system parameter is configured to permit plan reuse. The compiler is responsible for performing such "hard parse" steps (the details of which depend largely upon the particular DBMS

implementation) in the case of no match found or the value of the configurable system parameter disallows reuse.

[07] The systems and methods for sharing of execution plans for similar SQL statements reap many benefits, including: enhanced SQL processing performance without costly code revision, a decrease in the number of hard parse compilations that would otherwise be needed, and reduced CPU cycle time and latch contention, to name a few.

[08] Further details of aspects, objects, and advantages of the invention are described in the detailed description, drawings, and claims.

Brief Description of the Drawings

[09] Figure 1 is a flow diagram representing an example process for compiling a cursor according to one embodiment.

[10] Figure 2 is a flow diagram representing an example process for the soft parse methodology for avoiding compilation according to one embodiment.

[11] Figure 3 is a table representing various SQL statements with different combinations and positions of literals, showing varying levels of acceptability for sharing cursors for these statements according to one embodiment.

[12] Figure 4 is a flow diagram representing an example process for sharing of execution plans according to one embodiment.

[13] Figure 5 is a block diagram representing an example system for sharing of execution plans according to one embodiment.

[14] Figure 6 is a block diagram of a computer system that can be used in the implementation of one embodiment of the systems and methods for sharing of execution plans for similar SQL statements.

[15] Figure 7 is a block diagram of a two-tier client/server system that can be used to implement the systems and methods for sharing of execution plans for similar SQL statements according to one embodiment.

Detailed Description of the Preferred Embodiments

[16] The first time a SQL statement is processed by a server, it is usually compiled. Compilation is a multi-stage process beginning with a parse and ending with an execution plan as further described below. The compilation process is often referred to as a "hard parse" due to the sheer amount of work (i.e., in the form of function calls, routine invocations, memory accesses, etc.) involved to accomplish the task.

[17] Figure 1 is a flow diagram illustrating the basic SQL statement compilation steps according to an embodiment. In this embodiment, the database statement is a SQL statement compilation begins with parse phase 150. The parse phase is so named because the SQL statement is analyzed and parsed clause by clause into its constituent components creating an expression tree (sometimes called a parse tree). The expression tree is effectively the SQL statement mapped to a new data structure and is eventually what gets traversed later during execution. The parse phase can consist of steps 105-120, depending on the SQL implementation.

[18] The parse phase involves syntactical analysis, step 105, where the statement is analyzed for correct syntax, followed by step 110, where among other things, a

determination is made whether the referenced objects exist. In step 115, user permissions are analyzed by the semantic analyzer to determine if the client holds access privileges to the specific objects referenced in the SQL text. Finally, the parse phase concludes by developing an expression tree for the SQL text in step 120.

[19] The type checking stage 125 engages data type resolution between a client process and a server process, which verifies and corrects data type incompatibilities that can exist, for example, in a heterogeneous enterprise client/server network. For example, a user process (client) running on Windows NT against an Oracle Corporation ("Oracle") database running on MVS (multiple virtual storage) would require data type resolution between ASCII and EBCDIC.

[20] An objective of SQL statement compilation is the development of an execution plan in step 130. The execution plan is the result of optimization by an optimizer running as a separate process on the server. The optimizer accepts a parsed and analyzed statement from parse phase 150 and figures out the best method of execution for the particular statement based on a number of criteria, including: statistical information gathered during syntactical analysis (step 110) and semantic analysis (step 115); selectivity of the statement if the statement happens to be a DML (data manipulation language) statement; and optimization methodologies, such as cost-based optimization (CBO) and/or activity based optimization (ABO). The result of a hard parse is a memory-resident data structure, which dictates to the server, by way of the execution plan, the best method for carrying out the database statement request. A cursor is one example of such a data structure that is essentially a handle to a memory location where the details and

results of a parsed and optimized database statement reside. The cursor comprises, among other things, a compiled SQL statement with its associated execution plan. A cursor in this context resides in the server's shared memory pool and as such, it must be distinguished from the use of the term cursor from other data processing contexts, such as client-side procedural language originated cursors used to aid data table processing.

[21] The SQL statement compilation process described above (and the cursor resultant therefrom) are both non-optimal in many respects. From an efficiency standpoint, a hard parse is slow and cumbersome, often causing noticeable execution delays for the application end-user. Further, economies such as would be derived from a simple check for a pre-existing cursor that could be reused or shared are unrealized. For example, a simple scan of the shared memory pool before initiating a hard parse can uncover the existence of a pre-built cursor suitable for reuse by the existing SQL statement held in the shared memory pool.

[22] The systems and methods for sharing of execution plans for similar SQL statements overcome the disadvantages discussed above by reusing the execution plan of an existing cursor in those situations where a client issues a SQL statement similar to another SQL statement previously compiled. This can significantly reduce the expense of compilation involved in processing SQL statements. Expense of compilation can be measured in terms of expended CPU cycles and cache hits, among other things. Therefore, the systems and methods for sharing of execution plans for similar SQL statements have as one goal the minimization of these and other measures. Figure 2 modifies the hard parsing methodology of figure 1 to incorporate a search stage. Step

205 computes a hash value to determine if a matching SQL statement exists in the shared memory pool. If a match is found, step 210, then the performance penalty of initiating a hard parse can be avoided altogether—the existing plan compiled for the matching SQL statement will be reused for the instant SQL statement (step 220). If a match is not found, then compilation proceeds as in figure 1 (step 215), beginning with parse phase 150 and ending with execution plan creation 130. In the case of no match, no savings can be achieved. This process of searching for a SQL statement match is called a soft parse. In general, a soft parse is preferred over a hard parse because a soft parse consumes far fewer resources.

[23] One technique for avoiding hard parse compilation of a SQL statement is to share or reuse cursors for identical SQL statements. In this approach, the matching step 210 of Fig. 2 is accomplished if an existing cursor corresponds to a first SQL statement that exactly matches the SQL statement presently being processed. If so, a soft parse is performed, and then the execution plan of the existing cursor is shared/re-used for the current SQL statement to avoid performing a hard parse compilation.

[24] According to one embodiment of the invention, even if two SQL statement do not exactly match, a soft parse can often be performed if the two SQL statements differ by only one or more literals (e.g., constants). Thus, two statements are considered "matching" in this embodiment at step 210 if no more than the values of one or more literals in the SQL statements are different. One approach to identifying multiple SQL statements that only differ by literal values is to replace each literal within a SQL statement with a "placeholder" element. If the multiple similar SQL statements differ by

only one or more literals, then after the literals are replaced with common placeholder elements, each transformed SQL statement should match. Any suitable element can be employed as a placeholder. For example, in one embodiment, bind variables are used as placeholder elements.

[25] To illustrate, consider a first SQL statement that had previously been hard-parsed, in which the WHERE clause of the statement includes the constant "100" as follows:

```
SELECT * FROM EMP_T
    WHERE EMPLOYEE_ID = 100
```

This statement can be transformed by replacing the literal "100" with the bind variable ":X":

```
SELECT * FROM EMP_T
    WHERE EMPLOYEE_ID = :X
```

Now assume that a second SQL statement is subsequently issued as follows:

```
SELECT * FROM EMP_T
    WHERE EMPLOYEE_ID = 200
```

It can be seen that this second SQL statement differs from the first SQL statement only by the value of the literal in the WHERE clause. This second SQL statement can be transformed by replacing the literal with the same bind variable ":X". The following shows the transformed version of the second SQL statement:

```
SELECT * FROM EMP_T
    WHERE EMPLOYEE_ID = :X
```

A search of existing cursors will readily detect that the transformed version of the second SQL statement exactly matches the transformed SQL statement corresponding to the cursor for the first SQL statement. Only a soft parse is needed in this example to share/re-use the existing cursor that has already been created for the first SQL statement, thereby avoiding the expense of performing a hard parse compilation.

[26] During compilation, the optimizer tailors the execution plan of a SQL statement based in part on the literal values (constants) input by the user or the application when the statement was issued or written. A compiled SQL statement (in the form of a cursor) is specific to that statement, right down to the constants. In other words, what constitutes an optimal plan for one statement may not necessarily yield optimal execution for another statement that differs only by a constant. In developing an appropriate execution plan, the specific position of the literal within the statement may matter, as may the specific value of the literal vis-à-vis other object information, such as the number of rows in a table or view. As such, it is possible that cursor sharing is not desirable or usable for all types of SQL statements. In effect, certain types of SQL statements are more amenable to cursor sharing than other types of SQL statements. For example, there may exist certain types of SQL statements that should not be used for cursor sharing unless the statements are identically matched, while there may also exist other types of SQL statements that can be used for cursor sharing even if the statements are not exactly identical (e.g., literal-replacement is used to find a match).

[27] TABLE 1 (Fig. 3) sets forth examples of various SQL statements with different combinations and positions of literals, showing varying levels of acceptability for sharing

cursors for these statements according to one embodiment of the invention. Consider the first sample statement pair shown in TABLE 1 (i.e., statements 1 and 2). Each of these SQL statements insert a set of values ("543, JOE, 32" and "297, SALLY, 45") into a row of the same table ("EMP_T"). Thus, both statements are identical, except for the value of the literals in these statements. Based upon the particular structure and composition of these statements, it is likely an optimizer would produce the same or very similar execution plan for both statements.

[28] Consider the second sample statement pair shown in TABLE 1 (statements 3 and 4). Here, even though the two statements differ only by their literal values, an optimizer may build an execution plan for these two similar statements differently, depending upon various factors, including: the different join algorithms available, the existence of an index for one or more columns in either the EMP_T or DEPT_T tables, and access to data distribution statistics on the EMPLOYEE_ID column. For example, the optimizer may prepare an execution plan for the first of these statements that calls for a nested loop join, while the second statement may call for a hash join. It may be difficult or impossible by examining the statements' syntax alone to identify whether their plans will differ. It is the factors mentioned above, and not merely the syntactical concurrence of any two statements, that controls similarity.

[29] Thus, while the execution plan prepared for the first statement may still be usable by the second statement to produce a valid result, it is also likely that the execution plan prepared for the first statement will not be properly optimized for the second statement and will therefore be overly expensive to execute. For this reason, statements 3 and 4 of

TABLE 1 can be qualified as "suboptimally" shareable statements because although they only differ by a literal value, they likely have very different execution plans. Similarly, for ease of illustration, statements 1 and 2 of TABLE 1 can be qualified as "optimally" shareable because they are likely to have the same or very similar execution plans, without significant sacrifice to performance when sharing cursors because of the nature of the "INSERT INTO" clause and the placement of literals therein.

[30] Thus, the difference between a statement which can and ought to share a cursor with another similar statement, and a statement which can but may not want to share a cursor with another similar statement is now apparent. For the first statement pair of TABLE 1 (statements 1 and 2), sharing a cursor will not significantly impact performance. However, for the second statement pair of TABLE 1 (statements 3 and 4), sharing a cursor may affect performance depending on whether the respective execution plans for the two statements differ. If statements 3 and 4 of TABLE 1 do in fact have identical optimal plans, then sharing cursors is desirable. If the two statements do not have identical optimal plans, then an application or user may want to avoid cursor reuse.

[31] The systems and methods for sharing of execution plans provide a solution that allows for such control by the application designer or user as further explained below. In one embodiment, the systems and methods for sharing of execution plans are implemented as a dynamic parameter accepting one of three values: FORCE, SIMILAR, and EXACT. Consider a parameter called cursor_sharing, which stores the value. The following paragraphs explain the meaning of these three parameter values.

[32] EXACT

[33] cursor_sharing=EXACT is the default setting. For cursor_sharing set to EXACT, the system must find an exact match between the instant SQL text and a cursor in the shared memory pool in order for a cursor to be reused. An exact match means literal for literal; thus, only textually identical statements will invoke plan sharing. As such, this setting conforms to the behavior expounded above wherein the first pair of SQL statements of TABLE 1 would not share a cursor because they are not identical, literal for literal. The benefits of the systems and methods for sharing of execution plans are not realized with this parameter setting.

[34] SIMILAR

[35] For cursor_sharing set to SIMILAR, only similar statements that are optimally shareable will share an execution plan. For example, with this setting of the cursor_sharing parameter the first sample statement pair of TABLE 1 above would share a cursor, but the second sample statement pair would not.

[36] FORCE

[37] For cursor_sharing set to the value FORCE, execution plans will be shared among suboptimally shareable statements as well as among optimally shareable statements. Forcing plan sharing among suboptimally shareable statements may cause a potential hit to statement execution performance because FORCE trumps what would have been the optimal execution plan derived by the optimizer for that statement. Hence, cursor_sharing should only be set to FORCE when the benefits of cursor sharing are believed or known by the user or application designer to outweigh the consequences of suboptimal statement execution.

[38] Cursor sharing using the cursor_sharing embodiment can be effected in at least three ways according to an embodiment: using a command-line parameter to override the current setting, a server initialization parameter issued at system start-up, or a session-based parameter having the duration of only a single session. For example, the syntax of the ALTER SYSTEM and ALTER SESSION commands (e.g., in the Oracle 9i database system available from Oracle Corporation of Redwood Shores, California) can be modified to allow a cursor sharing parameter to be set as follows:

ALTER SYSTEM SET cursor_sharing={FORCE|SIMILAR|EXACT}

ALTER SESSION SET cursor_sharing={FORCE|SIMILAR|EXACT}

Also, information about the current setting of the cursor sharing parameter can be obtained by consulting one or more metadata views as is well known in the art and easily accomplished via a simple SQL run-time query against the appropriate view. Bind variable metadata can be obtained in a similar manner. For instance, a DBA who wishes to access information about internal bind variables (introduced below) can issue a SQL query against the Oracle system metadata view V$SQL_BIND_DATA using the following syntax:

SELECT * FROM V$SQL_BIND_DATA

WHERE BITAND(SHARED_FLAG2,256)=256

[39] Other implementation features can include SQL command-line parameters readable by the optimizer for performing one-time overrides of the current system parameter configuration. One such command-line construct is the hint provided by

Oracle. A user-supplied statement-level hint properly recognized by an Oracle optimizer can, for example, come as the following:

```
SELECT /*+CURSOR_SHARING_EXACT*/ * FROM EMP_T, DEPT_T WHERE
    (EMP_T.EMPLOYEE_ID<100) AND
    (EMP_T.ENAME=DEPT_T.ENAME)
```

[40] In one embodiment, a pair of similar statements is considered non-shareable if sharing a cursor compiled for one of the statements would produce incorrect results if shared by the other. For example, the third pair of similar statements of TABLE 1 are non-shareable because their differing literal values occur within an ORDER BY clause; reuse by statement 5 of an execution plan compiled for statement 6 could produce an incorrect result. To circumvent erroneous results caused by inappropriate cursor reuse, statements like these that differ by a literal in an ORDER BY clause are flagged as non-shareable so that the systems and methods for sharing of execution plans will not permit execution plan sharing in this case. In one embodiment, the systems and methods for sharing of execution plans prevent inappropriate cursor sharing by maintaining a list of SQL clauses for which cursor sharing is forbidden. Cursor sharing between the second pair of statements in TABLE 1, by contrast, would produce at least correct results for both statements despite a potential performance penalty. Non-shareable statements do not share a cursor unless the complete syntax of the statements is textually identical.

[41] Figure 4 is a user-level system flow diagram exemplifying the steps of the systems and methods for sharing of execution plans for similar SQL statements according to one embodiment. The process begins when the database server receives a SQL text from a database client in step 405, for instance, as an ad-hoc query or as a DDL (data

definition language) or DML request bundled within an OLTP (on-line transaction processing) application. In step 410 the server searches the shared system cache for an existing cursor compiled from matching SQL text. The shared system cache can be, for example, the shared SQL area of the Oracle shared pool. If an exact match is found, step 415, then the matching cursor is executed in lieu of compiling a cursor from scratch for the SQL text. If cursor_sharing=EXACT, step 440, then a cursor will be compiled from scratch for the SQL text (450) even though the statements might only differ by one or more literal values. In step 420, the server searches the system cache (shared pool) for a cursor compiled from similar SQL text. Two SQL texts are similar if they differ only by a literal. If a statement compiled from similar SQL text is not found, step 425, then a cursor will be compiled from scratch for that statement (step 450). If a similar statement is found in step 425, then processing continues with decision block 430.

[42] In decision block 430, the systems and methods for execution plan sharing first analyze whether the SQL text of the two similar statements is non-shareable. For example, if the two SQL texts are identical in all but the literals contained in an ORDER BY clause, then the statements are considered non-shareable and the systems and methods for execution plan sharing will not allow the cursor found in step 425 to be shared. A new cursor for the SQL text will then be compiled (450). Only if the two statements happen to be both non-shareable and identical, would decision block 415 permit cursor sharing to take place. In order words, non-shareable SQL statements do not share an execution plan unless the statements are textually identical.

[43] In step 435, the server determines whether the similar SQL statements are sub suboptimally shareable. If the statements are suboptimally shareable, such as the second pair of statements of TABLE 1, then if the cursor_sharing parameter is set to FORCE (445), the cursor will be executed for the SQL text. If the statements are suboptimally shareable but cursor_sharing is not set to FORCE, then a new cursor will be compiled for that SQL text (450).

[44] In one embodiment, the systems and methods for sharing of execution plans implement reuse by automatically replacing every SQL text literal with a bind variable. A bind variable is a canonical naming scheme whereby literal SQL text is substituted on a position by position basis with a variable having the format of a text string preceded by a colon, as in the following example:

```
SELECT * FROM EMP_T, DEPT_T WHERE
        (EMP_T.EMPLOYEE_ID<:X) AND
        (EMP_T.ENAME=DEPT_T.ENAME)
```

Automatic bind variable substitution occurs prior to the search in the shared pool for similar SQL text in step 420, if no exact SQL text match in step 410 is found. A bind variable substituted automatically by the systems and methods for sharing of execution plans is called an internal bind variable—distinguishable from user-coded bind variables. Despite the difference in terminology, system behavior is the same with respect to user bind variables and internal bind variables alike. Finally, automatic bind variable substitution is unconditional, and occurs whether a bind variable is replacing a safe or unsafe literal, terms which are defined and described below.

[45] Bind variables in general promote greater cursor reuse and reduce contention on the shared memory pool. A savvy developer will write his or her applications with this in mind, making use of bind variables wherever feasible. However, not all developers use bind variables consistently; as a result SQL processing can run suboptimally. The work-around for many DBAs is often to hire an consultant to either rewrite the code at considerable expense to the organization or to attempt performance fine-tuning by modifying select server parameters that may or may not directly effect SQL processing. The systems and methods for sharing of execution plans offer direct solution to optimizing SQL performance by providing a parameter whose advantages include promoting execution plan reuse without any code rework. Specifically, the systems and methods for sharing of execution plans have as an objective the improvement of SQL processing performance for applications that do not use bind variables or use them inconsistently.

[46] An improvement offered by the methods and systems for sharing of execution plans is a forced, or automatic replacement of each literal with a bind variable. Prior methods and systems relied on the user or application developer to undertake variable substitution as a remedial performance boost—substitution did not happen automatically. In other words, the user took responsibility for poor SQL processing performance, not the server. Alternatively, to avoid a costly rewrite, a user or DBA faced with a SQL processing problem awaited the next version of the application after the developer built and distributed a new release.

[47] According to one embodiment, if the system encounters a user-coded bind variable in a SQL text during automatic bind variable substitution, the system will ignore the variable completely; user-supplied bind variables go unacknowledged and unmodified. The classification of a literal as either safe or unsafe does, however, affect the manner of bind variable substitution. The designation of a literal as either safe or unsafe in this implementation parallels the distinction made between optimally shareable and suboptimally shareable SQL statements discussed earlier. A safe literal is one where substitution by another literal in the SQL statement where that literal resides would produce a new SQL statement having the same execution plan as before the substitution. If not the case above, the literal will be classified as unsafe. Thus, a suboptimally shareable SQL statement can also be characterized as one which differs by an unsafe literal from another similar SQL statement. Even for those SQL statements differing, for example, only by literals in an ORDER BY clause, bind variable substitution takes place. Such literals act more like keyword references to other parts or expressions of the SQL statement rather than to data values; hence, these literals are termed non-data literals. Nevertheless, as an implementation detail, the systems and methods for sharing of execution plans performs strictly unconditional bind variable substitution in this implementation. That is, , bind variable substitution proceeds in this implementation without regard to the safe, unsafe, or non-data status of the SQL text literals.

[48] Figure 5 is a block diagram of the components comprising the systems and methods for sharing of execution plans and showing the structural elements of the invention according to one embodiment. The database server comprises a compiler, a

cursor sharing monitor, and a search engine, each of which interact with the shared memory pool. The shared memory pool is the portion of system cache where open cursors are manipulated during SQL processing. The shared memory pool can be, for example, the database shared pool area.

[49] The database server responds to incoming client requests in the form of SQL statements and controls cooperative interaction among the database server modules. The search engine module manages the functionality of blocks 410 and 420 of figure 4 by checking the shared memory pool for both matching SQL text and similar SQL text, verifying the results, and interacting with the cursor sharing monitor and the compiler to further facilitate SQL processing.

[50] The cursor sharing monitor is invoked any time execution plans are to be shared among SQL texts. For instance, if block 410 determines there to be a matching SQL text in the shared memory pool, then execution plan reuse is monitored and controlled, step 455, by the cursor sharing monitor.

[51] By contrast, the compiler is invoked whenever execution plans are not to be shared among SQL texts. In other words, one compiler responsibility is the hard parse compilation process disclosed in figure 1. An example of a situation where cursor sharing is forbidden is when the system encounters two non-shareable statements, as in condition block 430. If a similar SQL text would return incorrect results under shared execution, then a cursor will be freshly compiled and run for that particular SQL text, step 450.

[52] Figure 6 is a block diagram of a computer system 600 upon which the systems and methods for sharing of execution plans for similar SQL statements can be implemented. Computer system 600 includes a bus 601 or other communication mechanism for communicating information, and a processor 602 coupled with bus 601 for processing information. Computer system 600 further comprises a random access memory (RAM) or other dynamic storage device 604 (referred to as main memory), coupled to bus 601 for storing information and instructions to be executed by processor 602. Main memory 604 can also be used for storing temporary variables or other intermediate information during execution of instructions by processor 602. Computer system 600 also comprises a read only memory (ROM) and/or other static storage device 606 coupled to bus 601 for storing static information and instructions for processor 602. Data storage device 607, for storing information and instructions, is connected to bus 601.

[53] A data storage device 607 such as a magnetic disk or optical disk and its corresponding disk drive can be coupled to computer system 600. Computer system 600 can also be coupled via bus 601 to a display device 621, such as a cathode ray tube (CRT), for displaying information to a computer user. Computer system 600 can further include a keyboard 622 and a pointer control 623, such as a mouse.

[54] The systems and methods for sharing of execution plans for similar SQL statements can be deployed on computer system 600 in a stand-alone environment or in a client/server network having multiple computer systems 600 connected over a local area network (LAN) or a wide area network (WAN). Figure 7 is a simplified block diagram

of a two-tiered client/server system upon which the systems and methods for sharing of execution plans for similar SQL statements can be deployed. Each of client computer systems 705 can be connected to the database server via connectivity infrastructure that employs one or more LAN standard network protocols (i.e., Ethernet, FDDI, IEEE 802.11) and/or one or more public or private WAN standard networks (i.e., Frame Relay, ATM, DSL, T1) to connect to a database server running DBMS 715 against data store 720. DBMS 715 can be, for example, an Oracle RDBMS such as ORACLE 9i. Data store 720 can be, for example any data store or warehouse that is supported by DBMS 715. The systems and methods for sharing of execution plans for similar SQL statements are scalable to any size, from simple stand-alone operations to distributed, enterprise-wide multi-terabyte applications.

[55] In one embodiment the system and methods for sharing of execution plans for similar SQL statements is performed by computer system 600 in response to processor 602 executing sequences of instructions contained in memory 604. Such instructions can be read into memory 604 from another computer-readable medium, such as data storage device 607. Execution of the sequences of instructions contained in memory 604 causes processor 602 to perform the process steps that will be described hereinafter. In alternative embodiments, hard-wired circuitry can be used in place of or in combination with software instructions to implement the present invention. Thus, the methods and system for sharing of execution plans for similar SQL statements is not limited to any specific combination of hardware circuitry and software.

[56] The systems and methods for sharing of execution plans facilitate greater user control over execution plan reuse than earlier systems and methods permitted. Furthermore, by implementing the systems and methods for sharing of execution plans as a user-programmable server parameter, the user is able to control cursor reuse without rewriting application code. In an embodiment whereby SQL text literals are replaced with internal bind variables, there is no noticeable impact on system performance vis-à-vis the performance had the SQL text been originally written with user bind variables.

[57] The systems and methods for sharing of execution plans for similar SQL statements derive many other important benefits. For one, forcing cursor sharing can significantly improve performance especially in situations where the application can tolerate suboptimal plans or as in an OLTP environment where suboptimal plans are not generally at issue on account of heavy reliance on indexes. Second, the anticipated increase in the number of parses (i.e., searches of memory for a matching SQL text) is offset by a decrease in hard parses performed (i.e., virgin compiles), greatly boosting performance in certain applications written with inconsistent use of bind variables. Third, measurable improvements in relative performance attributes, such as cycle time, memory usage, and latch contention make the systems and methods for sharing of execution plans desirable solutions having few, if any, side effects. Finally, for applications written at the outset using bind variables, the systems and methods for sharing of execution plans are designed to display no noticeable performance deterioration because the pre-check for a matching cursor in the shared memory pool will prevent a soft parse altogether, as a well-tuned application deserves.

[58] The systems and methods for sharing of execution plans can be implemented as a direct improvement over existing systems and methods for ad-hoc SQL processing, as described herein. However, the present invention contemplates as well the enhancement of other DBMS sub-systems and interfaces including, by way of example, necessary modifications to one or more proprietary procedural languages, such as Oracle PL/SQL, or code-level adjustments or add-ons to a proprietary or open-system architecture such as Java stored programs, needed to extend the functionality of the present invention.

[59] Other such modifications to existing program code needed to fully implement the present invention can also include enhancements to the algorithms employed by the various system modules of figure 5. For instance, the algorithms that the optimizer follows to build execution plans often involves the calculation of predicate selectivity. As a practical matter, the optimizer will require access to the SQL statements' proposed bindings on the first invocation of the compiler—that is, the first time a SQL text is issued from a client and compiled during a session. An implementation of the present invention whereby the optimizer has no prior knowledge of the bindings may result in performance degradation (as a result of a bad plan being built) that could exacerbate, rather than overcome, the performance problem that the methods and systems of execution plan sharing set out to solve. Thus, the optimizer would ideally need to be rewritten to permit it access to the bindings in advance of substitution. This and other similar code modifications may be necessary to a successful implementation and it is fully within the contemplation of the present invention that such modified or additional code be developed.